Subject:.....
Date:.....

String Matching

Finding all occurrence of a pattern in a given text

Application:

→ while using editors, word processors, browsers

→ login name & password checking

→ virus detection

→ Headers analysis and data communication

→ DNA sequence analysis.

problem:

→ we say that pattern P occurs with shift s in text T if:

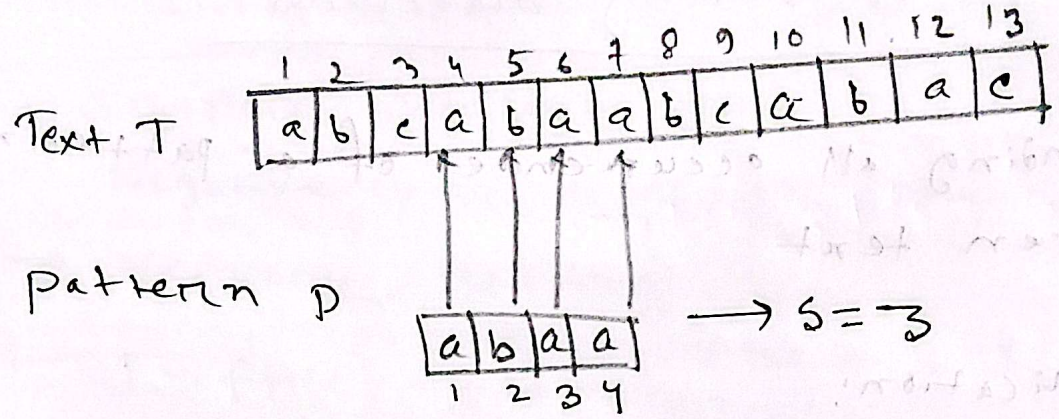
a) $0 \leq s \leq n-m$ and

b) $T[(s+1) \ldots (s+m)] = P[1 \ldots m]$

→ If P occurs with shift s in T, then s is a valid shift, otherwise s is an invalid shift

Subject:.....

Date:.....



shift $s = 3$ is valid shift

($n = 13$, $m = 4$ and $0 \leq s \leq n - m$ holds)

Naive String matching algorithm

Input: Text strings $T[1 \ldots n]$ and

$P[1 \ldots m]$

result: All valid shifts displayed

Pseudocode:

```
Naive-String-Matcher (T, P)
    n = length[T]
    m = length[P]
    for s = 0 to n-m
```

Subject:.....
Date:.....

if $P[1 \ldots m] == T[(s+1) \ldots (s+m)]$

print "pattern occurs with shift" s

Brute Force approach

The brute force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found.

Ex:

Two Roads divided the two point

↑
Roads

Two "

↑
Roads

Two "

↑
Roads

Two Roads "

↑
Roads

Two Roads }
Roads } match

Subject:.....

Date:.....

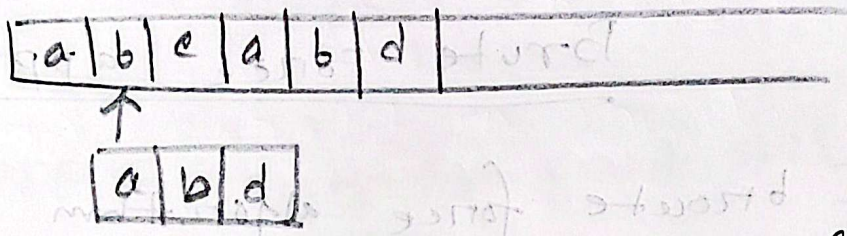
Ex: (01)

T = | a | b | c | a | b | d | a | a | b | c | d | e |

P : | a | b | d |

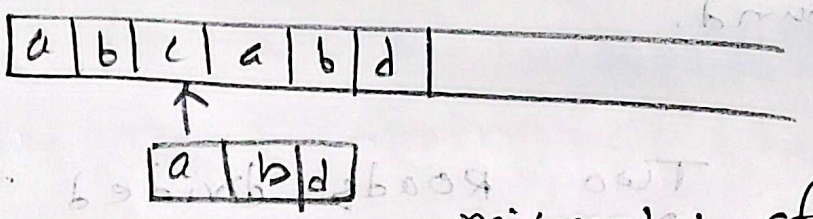
mismatch after 3 comparison

step-02 :



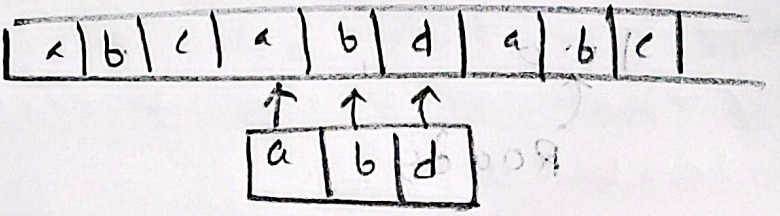
→ mismatch after 1 comparison.

step-03



mismatch after 1 compar

step-04



match found after 3 comparisons.

Thus after 8 comparison the substring p is found in T.

Subject:.....

Date:.....

problem of this algorithm

T = a a a a a . . . . a a f of size say "n"

p = a a a f of size 4

T: a a a a a ..... a a f

↑ ↑ ↑ ↑

p: a a a f

→ mismatch after 4 comparisons

T: a a a a a a a a ..... a a a f

↑ ↑ ↑ ↑

p: a a a f

→ mismatch after 4 comparison

This will continue to happen until (n-4)th alphabet in T is compared with the characters in p and thus the no. of comparisons required is (n-4)4+4

Subject: .....

Date: .....

worst case running time

→ at every step after "m" comparisons a mismatch will be found

→ these "m" comparison will be done for (n-m) characters in T.

Thus the running time obtained is

$$(n-m)m + m$$

~~$\therefore Tc = O(n)$~~

$\therefore Tc = O(M(N-M+1))$

Best case

If pattern found in First M position of text.

Ex: a a a a a a a a a H
a a a a a

→ after 5 comparison

total no. of comparison M

Subject:.....
Date:.....

if the total no. of comparison N

∴ Best case : O(N)

✓To reduce time complexity then come kmp algorithm

Kunath Morris Pratt algorithm

The kmp storing searching algorithm is differs from the brute force algorithm by keeping track of information gained from previous comparisons.

A failure function (f) is computed that indicates how much of the last comparison can be reused if it fails.

Subject:.....

Date:.....

Algorithm failure function (P):-

```
F[0] ← 0
i ← 1
j ← 0

while i < m
    if P[i] = P[j]
        {we have matched j+1 char}
        F[i] = j+1
        i++
        j++

    else if j > 0 then
        {use failure function to
         shift P}
        j = F[j-1]

    else
        F[i] ← 0 {no match}
        i++
```

Subject:.....

Date:.....

	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a

	0	1	2	3	4	5	6
	a	b	a	b	a	c	a
F[ ]	0						

j=0, i=1, m=2

	0	1	2	3	4	5	6
	a	b	a	b	a	c	a
F[ ]	0	0					

j=0, i=2

	0	1	2	3	4	5	6
	a	b	a	b	a	c	a
F[ ]	0	0	1				

j+1 = 0+1 = 1

0+1 = 2

j=1, i=3

	0	1	2	3	4	5	6
	a	b	a	b	a	c	a
	0	0	1	2			

j+1 = 1+1 = 2

j=2, i=4

	0	1	2	3	4	5	6
	a	b	a	b	a	c	a
	0	0	1	2	3		

j+1 = 2+1 = 3

j≥3, i=5

	0	1	2	3	4	5	6
	a	b	a	b	a	c	a
F[ ]	0	0	1	2	3		

j-1 = F[2] = 1

Subject: .....

Date: .....

0	1	2	3	4	5	6
a	b	a	b	a	c	a
0	0	1	2	3		

$j=0, i=5$

0	1	2	3	4	5	6
a	b	a	b	a	c	a
0	0	1	2	3	0	

$j=0, i=6$

0	1	2	3	4	5	6
a	b	a	b	a	c	a
0	0	1	2	3	0	1

$j=1, i=7$

terminate

Algorithm kmpmatch (T, P)

```
f ← failure function(P)
i ← 0
j ← 0
while i < n
    if T[i] = P[j]
        if j = m
            return i-j {match}
        else i++
             j++
```

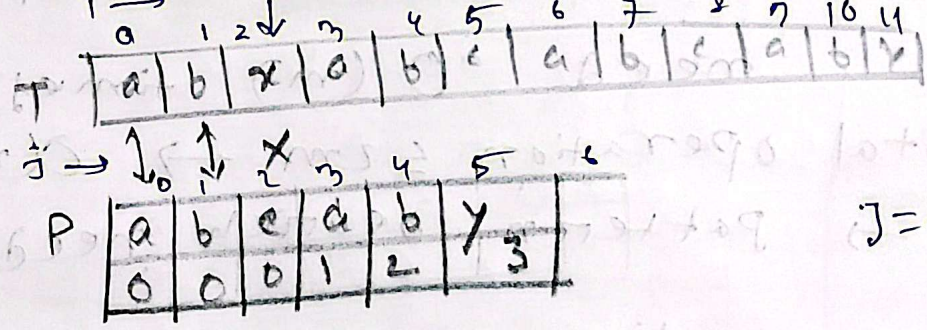
Subject:.....

Date:.....

```
else if i>0
        j ← F[j-1]
else
        i++
return -1 {no match}
```

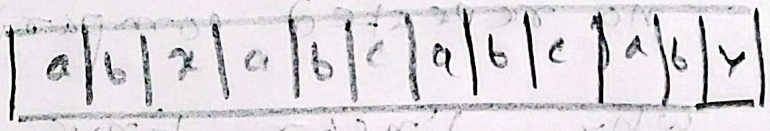
ex:

01

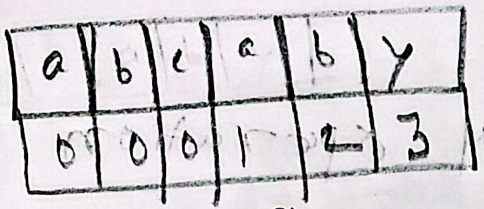


j=0, i=2

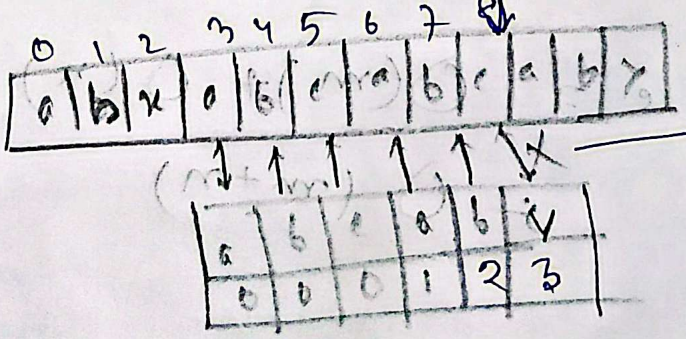
02



X — mismatch.



03

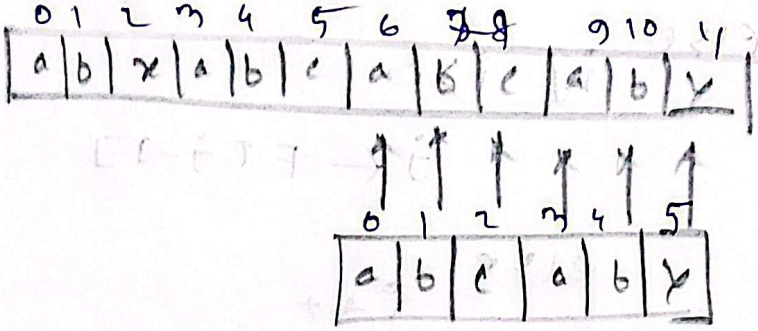


j=2, i=8

Subject:.....

Date :.....

(04)



complexity :

→ failure function or LPS creation need O(m) times.

total operation ≤ 2m → O(m)

→ pattern search need O(n) times. here no backtracking no repeated comparison. that's why linear time needed

∴ total operation ≤ 2n → O(n)

∴ Te = O(m) + O(n)

= O(m+n)

Subject:.....

Date:.....

Space: KMP store only one extra memory for LPS or failure function whose size m.

$\therefore$ SC = O(m).

Rabin Karp algorithm

Text: a a a a a b

1 2 3 4 5 6

n = 6

1+1+1 = 3

Pattern: a a b

1 2 3

m = 3

1+1+2 = 4 → H(P) hash code

a = 97 a = 1

b = b = 2

c = 3

d = 4

a a a a a b ⇒ a a a a a b

1+1+1 = 3 1+1+2 = 4 = P

Subject:.....

Date:.....

Text: a b c d a b c e

P: b c e

2+3+5 = 10

abc = 6

bcd = 9

cda = 8

dab = 2

bce = 10

So match : b c e

But it has drawback which is given bellow:—

~~T.C = O (n-m+1)~~

Text: c c a c c a a c d b a

P: d b a

4+2+1 = 7

cca = 7

cac = 7

acc = 7

cca = 7

here multiple text pattern size same as original pattern but not same as alphabet which is a drawback.

caa = 5

aac = 5

acd = 9

cdb = 9

dba = 7

Subject:.....
Date:.....

this is called spurious hits.

For this, approach maximum time needed : O(mn)

But for Rabin Karp algorithm time complexity : O(n-m+1)

which is reduce time compexity.

How to avoid spurious hits:

Spurious Hit: occurs, when the Hash value of the pattern matches the hash value of text substring, but the actual characters do not match.

Avoid spurious hits:-

use strong rolling Hash function.

c c a c c a a c d b a

Subject: 100+30+3

Date: 3×101

P: 1 2 3

m=3 d b a

$9\times10^2 + 2\times10^1 + 1\times10^0$

$= 400 + 20 + 1 = 421$

$P[1]\times10^{m-1} + P[2]\times10^{m-2} + P[3]\times10^{m-3}$

here, there is no spurious hits.

Only match dba Substring to the given pattern.

Now,

c c a = 331 (Not P)

c a c = 313

a c c = 133

c c a = 133

c a a = 131

a a c = 331

a c d = 124

c d b = 342

d b a = 421 (match)

200+3×0

100+20+4

200+40+1

400+20+1

Subject:.....

Date:.....

Rolling Hash Function of a string of length m

$$H = D^{m-1} \cdot p_0 + D^{m-2} \cdot p_1 + D^{m-3} \cdot p_2 + \dots + D^{m-m} \cdot p_{m-1}) \bmod q$$

Here, m = size of pattern

q = a prime number. (3, 5, 7 ...)

t = text substring character

sliding to next substring:

$$H_{new} = (d \cdot D \cdot (H_{old} - t_i \cdot D^{m-1}) + t_{i+m}) \bmod q$$

let,

A = 1
B = 2
C = 3
D = 4

Ex: T = "ABCD"

P = "ABC"

① For "ABC"

$$H_0 = (256^2 \cdot \underset{A}{1} + 256^{2-1} \cdot \underset{B}{2} + 256^0 \cdot \underset{C}{3}) \bmod 7$$

② For "BCD",

$$H_1 = \left(256 \left(H_0 - \underset{A}{1} \cdot 256^2\right) + \underset{D}{4}\right) \bmod 7$$

Subject:.....

Date:.....

advantage:

→ O(1) time to compute hash of next window

→ avoid recomputing hash of full substring

→ Enables fast substring search in Rabin—Karp

complexity:

Best case: O(n+m)

- only hash comparison
- No collisions
- very fast

Average case: O(n+m)

- Few collision
- Occasional character check

Subject:.....
Date:.....

Worst case: $O(mn)$

- Many Hash collison
- Every ~~window~~ substring need a full character comparisoin
- Become like brute force.

Space: $O(1)$

- only few variables (hash values, constants)

pseudocode:

~~RTP~~

```
Rabin-Karp(T, P)
  n = T.length, m = P.length
  hP = Hash(P[...])
  hT = Hash(T[...])
  for s = 0 to n-m
    if (hP = hT)
      if (P[0...m-1] = T[s+0...s+m-1])
        print s found
    if (s < n-m)
      hT = Has(T(s+1... s+m))
```

$q = 2^{32}$ → data type (sign bit) 32 bit

Subject:

Date:

Pseudocode:

```
Rabin-Karp-Matcher (T, P, d, q)
    n = T.length
    m = P.length
    h = d^(m-1) mod q
    p = 0
    t0 = 0
for i = 1 to m
    p = (dp + P[i]) mod q
    t0 = (d t0 + T[i]) mod q
for s = 0 to n-m
    if p = ts
        if P[1...m] = T[s+i ... s+m]
            print "pattern occurs
                with shift" s
    if s < n-m
        ts+1 = (d(ts - T[s+1]h) +
            T[s+m+1]) mod q.
```

Greedy Algorithm

Subject:.....

Date:.....

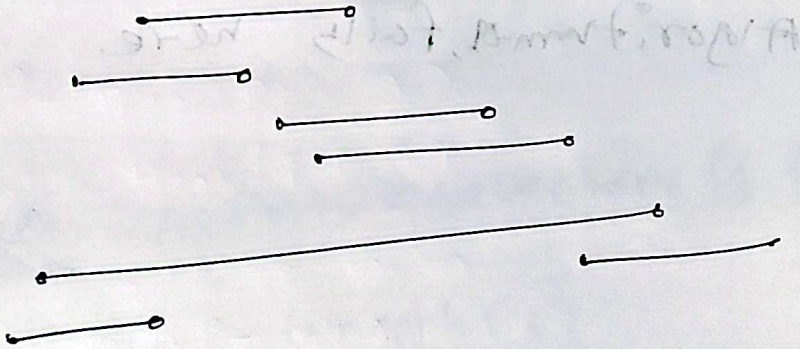
Activity selection problem

selecting the no. of non-overlapping activities based on their start and finish times using a greedy approach that picks the earliest finishing activity first.

Input: list of time interval L

Output: a non-~~ove~~ overlapping subset S of the intervals

maximize |S|)



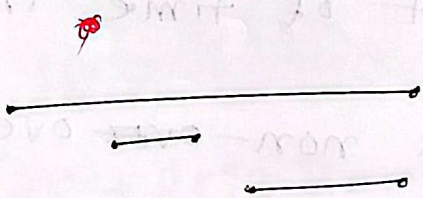
3,7
2,4
5,8
6,9
1,11
10,12
0,3

Subject:.....

Date :.....

Algorithm - 01

1. Sort the activities by the starting time
2. Pick the first activity "a"
3. Remove all activities conflicting with "a"
4. Repeat.



when choose as starting time ~~based on~~ the we cannot attend maximum times.

∴ Algorithm-01 fails here

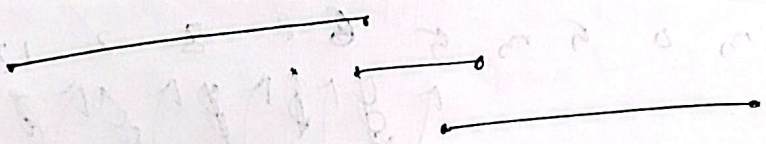
Subject:.....
Date :.....

Algorithm-03

→ sort the activities by ending time

→ pick the activity which ends first

→ remove all activities conflicting with a



for this case we find maximize result.

This algorithm-02 are optimal solution to the activity selection problem.

Pseudocode:

```
Greedy-Activity-Selector (s, f)
    n ← length[s]
    A ← {1}        // automatically select first activity
    j ← 1          // last activity selected so far.
    for i ← 2 to n
        if s[i] >= f[j] then
            A ← A ∪ {i}    // add activity i to the set
```

Subject:.....

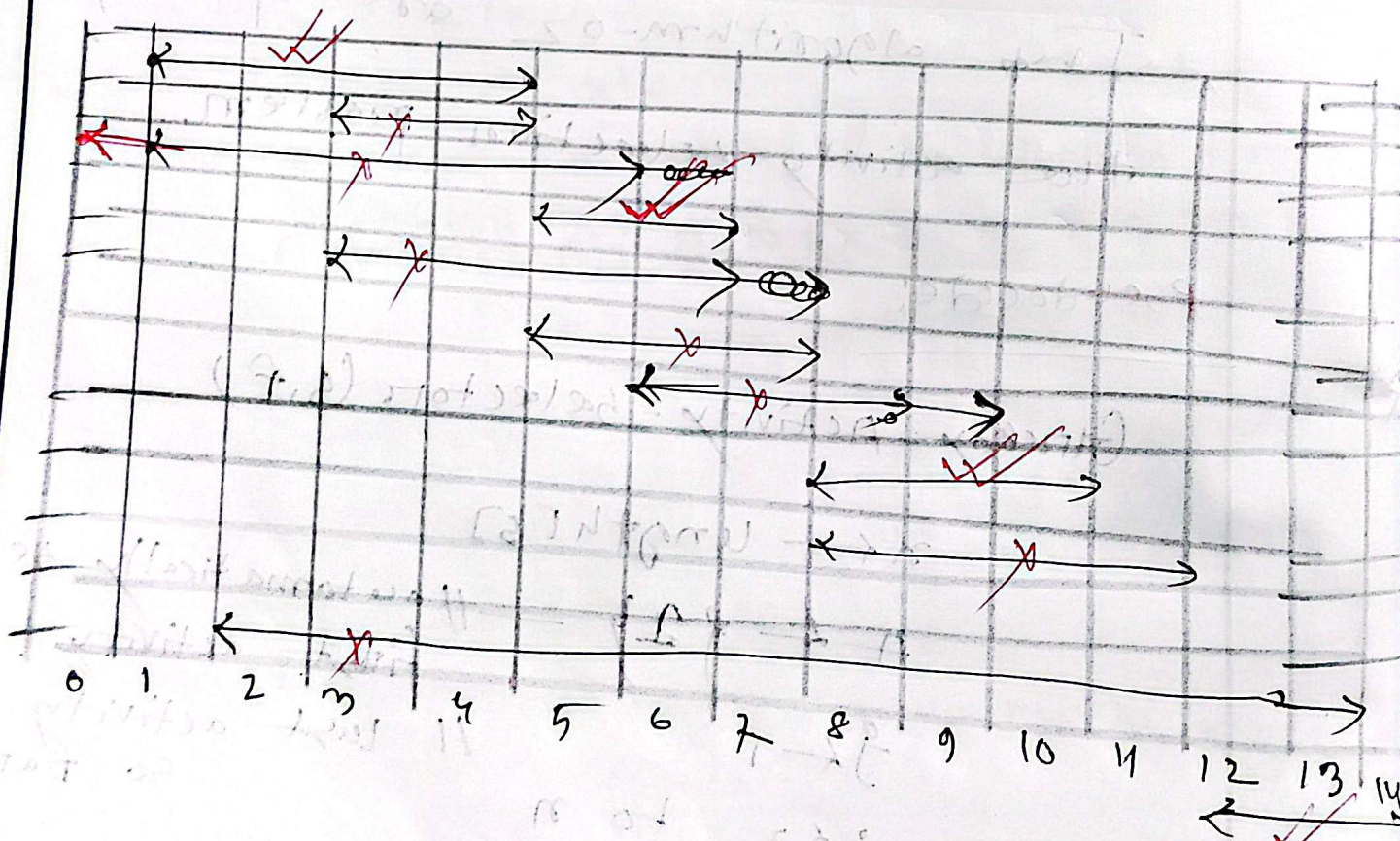
Date:.....

j ← i // record last activity add

return A.

Simulation:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Subject:

Date :

→ সবগুলোর মধ্যে Finishing time যেটা ছোট সেটা choose করবো

→ যেগুলো overlape করবে সেগুলো remove করবো

why this algorithm is optimal ?

we will show that this algorithm uses the following properties :—

→ The problem has the optimal substructure property

→ The algorithm satisfy the greedy choice property

→ thus it is optimal.

complexity:

For sorting total time needed $= O(n\log n)$

selecting activities using single loop, which takes times $O(n)$.

total: $O(n\log n) + O(n) \simeq O(n\log n)$

sc: $O(1)$

is a lossless data compression algorithm that assign shorter binary codes to more frequent characters and longer codes to less frequent characters

Subject:

Date:

Huffman coding - free code.

→ It used a greedy strategy to build an optimal prefix

→ widely used technique for data compression (20% to 90%)

→ assume the data to be a sequence of characters

→ looking for a effective way of storing the data.

→ Binary character code.

fixed-length codes

Ex: Data file 100,000 characters

	a	b	c	d	e	f
Frequency (thousand)	45	13	12	16	9	5

3 bits needed

a = 000, b = 001, c = 010, d = 011, e = 100, f = 101

requires: 100,000 * 3 = 300,000 bits

need more size, that's why comes Huffman codes.

Subject:.....

Date:.....

Huffman code

Here, Assign short codewords to frequent characters & long codewords to infrequent characters.

Ex: a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

∴ total = (45.1 + 13.3 + 12×3 + 16.3 + 9.4 + 5.4)×1000

= 224,000 bits.

where, 100,000 > 224,000 bits

—reduce bits, which is efficient.

Subject:.....

Date :.....

prefix code

→ codes for which no codeword is also a prefix of some other codeword

→ Better name " prefix-free ~~codewor~~ codes

we can achieve optimal data compression using prefix codes

Encoding with binary character code

concatenate the codewords

Ex: a=0, b=101, c=100, d=111, e=1101, f=1100

abc = 0.101.100 = 0101100

Subject:.....

Date:.....

Decoding with binary codes.

⇒ code to → characters

Ex: a=0, b=101, c=100, d=111, e=1101, f=1100

0010111101 = 0.0.101.1101 ϕ

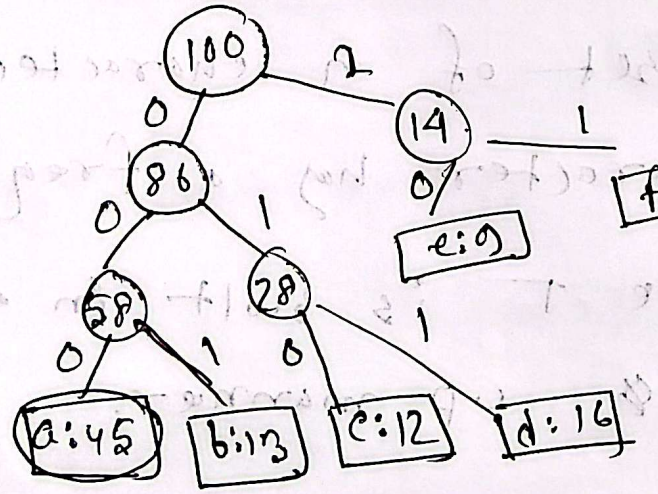
= aabe

prefix code representation :-

Binary tree whose leaves are the given characters.

0 → means "go to the left child"

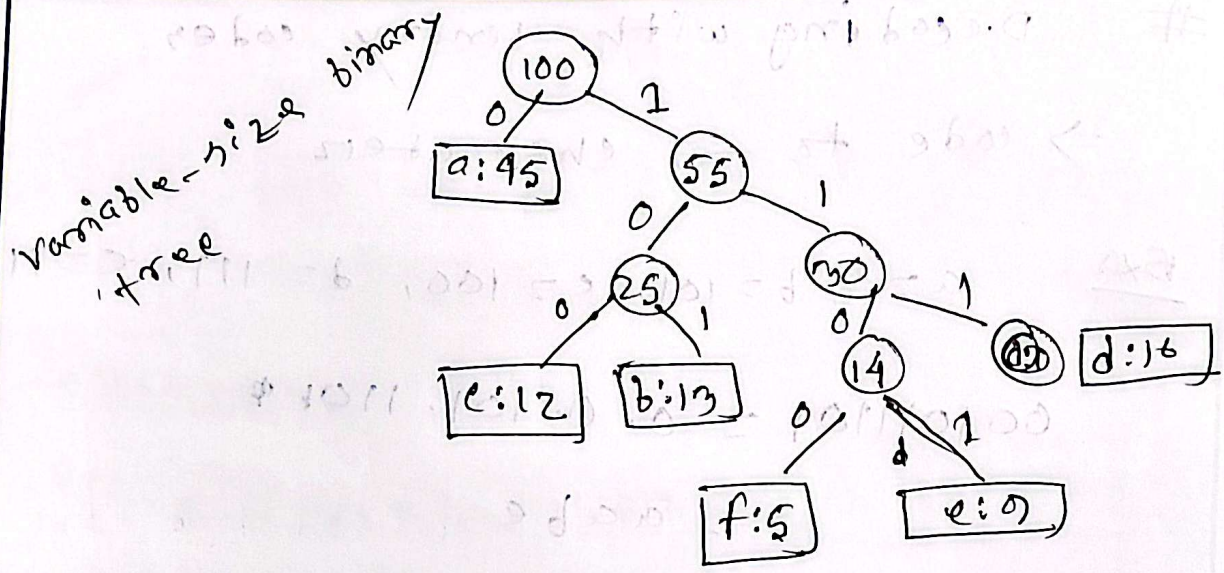
1 → " " " " right "



→ this tree for fixed-length which are expensive that's why we need variable-length string tree. to reduce cost.

Subject:.....

Date :.....



We construct a Huffman code

A greedy algorithm that constructs an optimal prefix code called a Huffman code.

Assume that,

→ C is the set of n characters

→ Each character has a frequency fre

– The tree T is built in a bottom up manner.

Subject:.....

Date:.....

Idea:

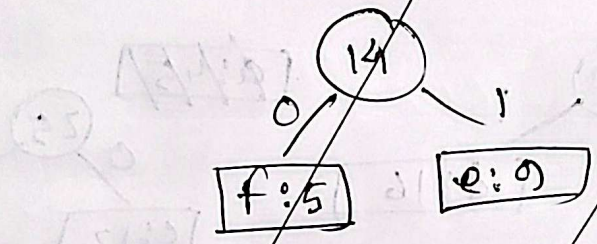
→ start with a set of |c| leaves

→ use min-~~heap~~ priority queue Q

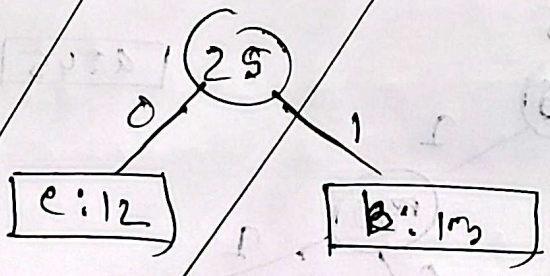
Here,

f:5	e:9	c:12	b:13	d:16	a:45

(1) let e and e node merge (9+5=14)



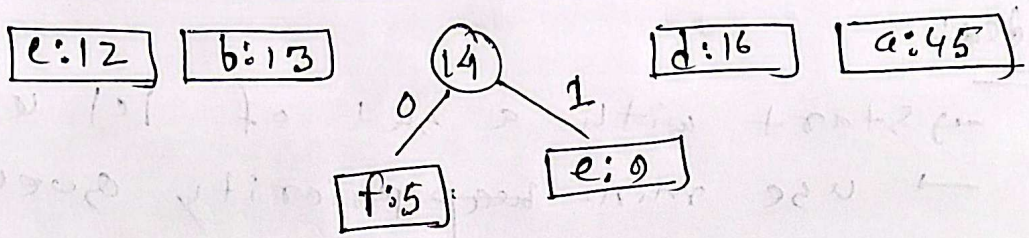
(2) take c:12 and b:13 node merge (12+13 =25)

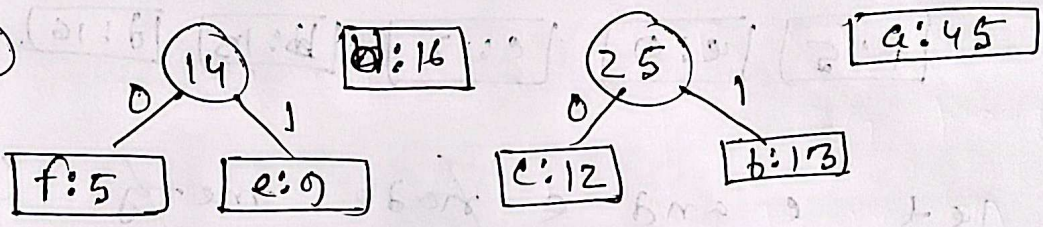


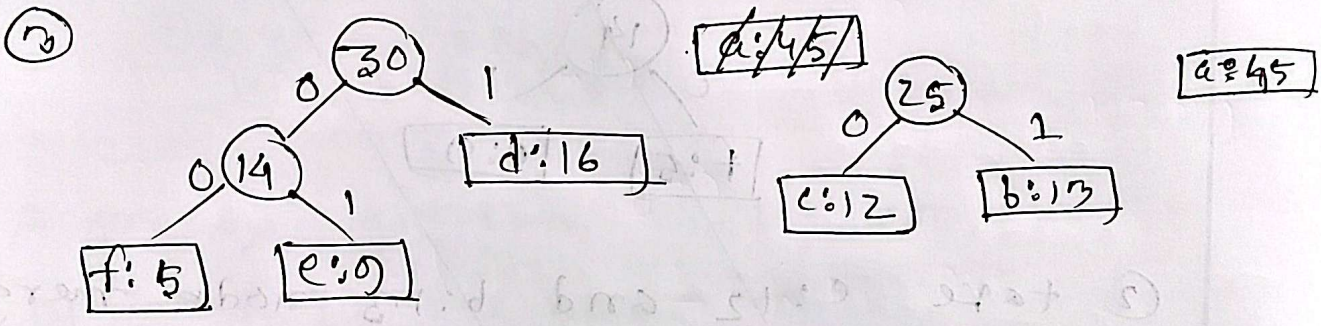
(n) now take d:16 and a:45 merge (61)

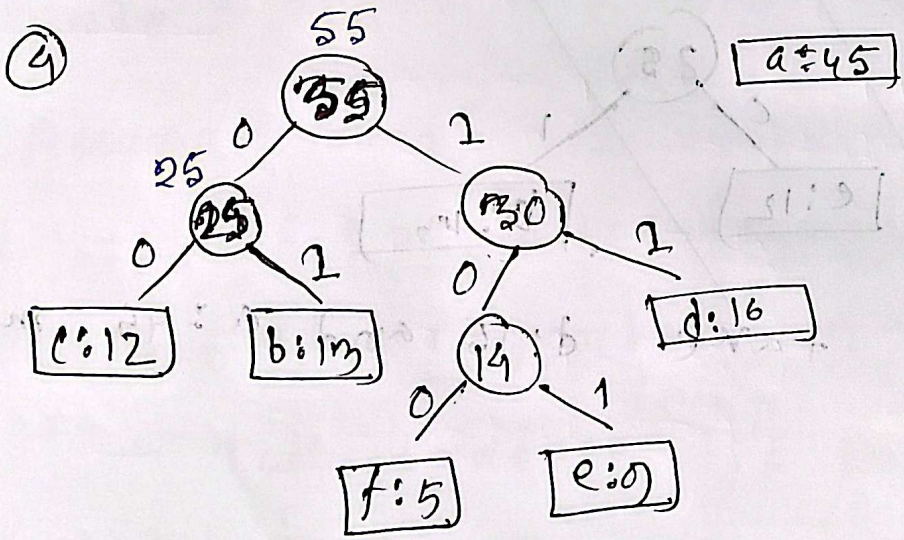
Subject:.....

Date :.....

① 

② 

③ 

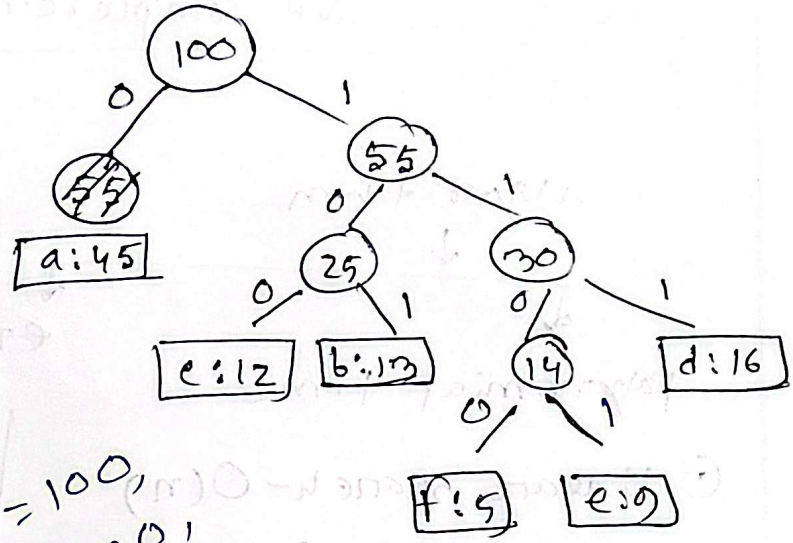
④ 

⑤

Subject:.....

Date:.....

(5)



code: a=0,
b=101, c=100,
d=111, f=1100,
e=1101.

——Done.

pseudocode:

Huffman(c)

n ← |c|

Q ← c,

for i ← 1 to n-1 — O(n)

do allocate a new node z

left[z] = x = Extract-min(Q)

right[z] = y = Extract-min(Q)

f[z] = f[x] + f[y]

Insert (Q, z).

n

O(n log n)

total Tc = O(n log n) heap

Sc = O(n) → for min